Fermi National Accelerator Laboratory

FERMILAB-TM-1742

A Note on the Automated Differentiation of Implicit Functions

L. Michelotti
Fermi National Accelerator Laboratory
P.O. Box 500
Batavia, Illinois 60510

June 1991

Operated by Universities Research Association Inc. under contract with the United States Department of Energy

A note on the automated differentiation of implicit functions.

Leo Michelotti

June 27, 1991

The question is this: Can automated differentiation be used on functions that are defined implicitly, recursively, or iteratively? Consider, for example, the simple function $x(m)$ defined implicitly by the equation,

$$x(m) = \cos(\, m \cdot x(m)\,) \tag{1}$$

Simple recursion (iteration?) can be used to construct $x(m)$ for m in the approximate range, $m \in (0, 1.2)$, determined by the condition $|m \sin(\, m \cdot x(m)\,)| < 1$. That is, the pseudocode,

```
x := 0;
read m;
until convergence do:  x ← cos(mx);
```

will converge on the value $x(m)$. *Will it converge on its DA prolongation?*[1] If it does, then we have a method for computing *exactly*, to machine accuracy, the derivatives of (at least some) implicit functions. If these functions define the critical points of a map, as this one does, then their derivatives can be used to study the dependence of these critical points on the map's parameters.

An experiment done using the C++ package MXYZPTLK [1] indicates that it does. The source code and output of this experiment, slightly edited for aesthetics, are given on the following pages. After preliminary boilerplate, we initialize a DA variables, `xd` and `md`, on SOURCE A line 12. A first look at `xd`, called for in SOURCE A line 19, is recorded in OUTPUT A lines 3-12. SOURCE A lines 21-23 call for 50 recursive steps (much more than necessary) of the recursion, after which SOURCE A lines 26 and 28 print out the variable and its difference from $\cos(\texttt{md} \cdot \texttt{xd})$. These are recorded in OUTPUT A lines 16-33. We see that `xd` has changed, indicating that a single pass was not enough. The *real* test, however, occurs after SOURCE A line 33, where comparisons are made between $x(m)$ as obtained by double-precision recursion and evaluating the polynomial represented by `xd`. As one

[1] A prolonged function explicitly carries information about derivatives. For terminology used in this memo see [1] and [2].

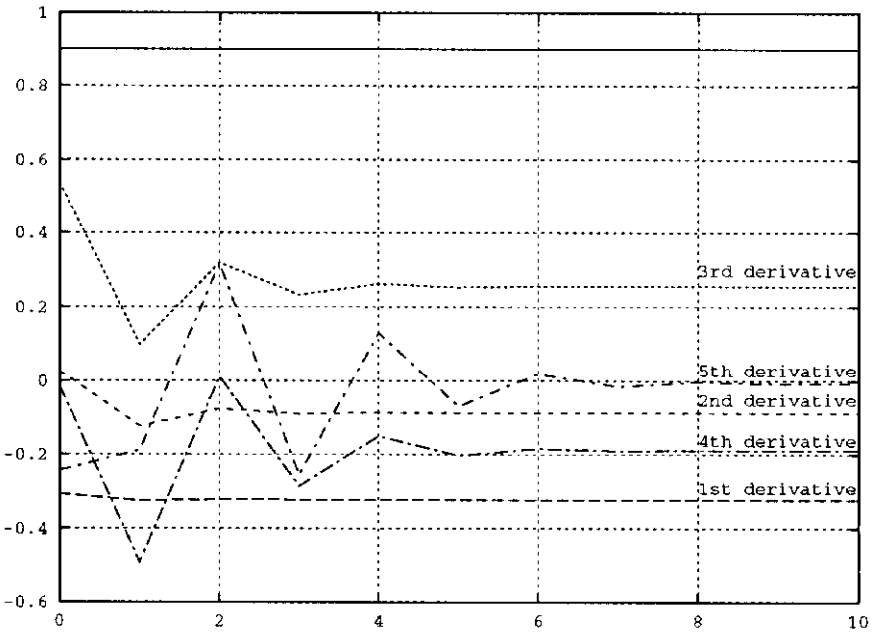

Figure 1: Behavior of the coefficients with iteration number.

can see from OUTPUT A lines 36ff, printed by SOURCE A lines 41-42, the agreement is "perfect."

Although 50 recursive steps were used in this test, the derivatives of $x(m)$ converge after the first ten or so, as illustrated in Figure 1. What are displayed in the figure are the *weighted* derivatives, the coefficients of the polynomial representation of $x(m)$ about $m = 0.5$.

As mentioned above, the range of m for which the recursion, Eq.(1), converges is very limited. We can extend our computation of $x(m)$ by using Newton's algorithm instead. SOURCE B is such a program. In order to calculate $x(m)$, we find a zero of the function, $F(x) = x - \cos mx$. This function is defined in SOURCE B lines 13-14, both in double precision and DA form. A Newton's solver is implemented in SOURCE B lines 16-45. The heart of this is contained in lines 32-33, where we construct a DA variable, `G`, that corresponds to a single Newton step, $x \leftarrow x - F/F'$. (The array `dfNdx` was defined in line 6.) Notice that as far as the solver is concerned F is an arbitrary function. `G` then invokes its double-precision **multiEval** method, in line 34, in order to set up the next recursive step of the loop (lines 28-40). (**multiEval** evaluates the polynomial represented by the DA variable.) The main progam invokes this solver at line 65 to construct $x(m)$ at one value of m. Then, in lines 70-75, `G` invokes its **multiEval** evaluation method *in DA mode* repeatedly in order to construct the derivatives of $x(m)$. (As before, doing this fifteen times was conservative overkill.) The resultant polynomial expansion is validated in lines 86-95, where $x(m)$ is calculated and tested for a range in m, the output of the program being

recorded in OUTPUT B. Notice the errors creeping into the fifth significant figure at the limits of this range, indicative of requiring a more than six-term polynomial. The number of derivatives computed was determined in SOURCE B lines 50 and 55 via the variable `maxWeight`. A subsequent run, using `maxWeight` = 9, produced complete agreement over the full range, and beyond.

We have demonstrated how DA can be used to calculate derivatives of a function defined implicitly. In the next note we shall extend these considerations to multi-dimensional maps, including the discussion of a tool for calculating fixed points from the source code of a mapping.

References

[1] Leo Michelotti. MXYZPTLK: A practical, user-friendly C++ implementation of differential algebra: User's guide. Fermi Note FN-535, Fermilab, January 31, 1990.

[2] Leo Michelotti. Differential algebras without differentials: an easy C++ implementation. In Floyd Bennett and Joyce Kopta, editors, *Proceedings of the 1989 IEEE Particle Accelerator Conference.* IEEE, March 20-23, 1989. IEEE Catalog Number 89CH2669-0.

C++ SOURCE A

```
 1 #include <stdio.h>
 2 #include "mxyzptlk.rsc"
 3
 4 main() {
 5 const double  tolerance = 1.0e-7;
 6 const int     dim       = 1;
 7 const int     maxWeight = 6;
 8 double        m, x, xOld;
 9 int           i;
10
11 DASetup( dim, maxWeight );
12 DA md, xd;
13
14 m = 0.5;
15 x = 0.900367;  // The value of  x( 0.5 ) ..
16 md.setVariable( m, 0 );
17 xd = cos( md * x );
18 printf( "\n\nAfter single step:\n" );
19 xd.peekAt();
20
21 for( i = 0; i < 50; i++ ) {
22   xd = cos( md * xd );
23 }
24
25 printf( "\n\nAfter the loop:\n" );
26 xd.peekAt();
27 printf( "\n\nComparison test:\n" );
28 ( xd - cos( md * xd ) ) .peekAt();
29
30 double mLo  = 0.4,
31        mHi  = 0.6,
32        step = 0.02;
33 printf( "\n\nThe real test:\n" );
34 for( m = mLo; m < (mHi + 0.5*step); m+=step ){
35   xOld = x;
36   x = cos( m*xOld);
37   while( ( fabs( x - xOld ) > tolerance ) ) {
38     xOld = x;
39     x = cos( m*xOld );
40   }
41   printf( "%lf  %lf  %lf\n",
42           m, x, xd.multiEval( &m )
43         );
44 }
45
46 } // End function: main()
```

OUTPUT A:

```
 1 After single step:
 2
 3 Count  = 7, Weight = 6, Max accurate weight = 6
 4 Reference point:
 5 5.000000e-01
 6 Index:  0  Value:  9.003673e-01
 7 Index:  1  Value: -3.917774e-01
 8 Index:  2  Value: -3.649462e-01
 9 Index:  3  Value:  5.293309e-02
10 Index:  4  Value:  2.465396e-02
11 Index:  5  Value: -2.145539e-03
12 Index:  6  Value: -6.662000e-04
13
14 After the loop:
15
16 Count  = 7, Weight = 6, Max accurate weight = 6
17 Reference point:
18 5.000000e-01
19 Index:  0  Value:  9.003672e-01
20 Index:  1  Value: -3.217713e-01
21 Index:  2  Value: -8.719257e-02
22 Index:  3  Value:  2.550414e-01
23 Index:  4  Value: -1.890445e-01
24 Index:  5  Value: -6.608513e-03
25 Index:  6  Value:  1.761623e-01
26
27 Comparison test:
28
29 Count  = 1, Weight = 0, Max accurate weight = 6
30 Reference point:
31 5.000000e-01
32 Weight: 0   Value: 0.000000e+00
33 Index:  0
34
35 The real test:
36 0.400000  0.931399  0.931399
37 0.420000  0.925413  0.925413
38 0.440000  0.919302  0.919302
39 0.460000  0.913082  0.913082
40 0.480000  0.906766  0.906766
41 0.500000  0.900367  0.900367
42 0.520000  0.893899  0.893899
43 0.540000  0.887373  0.887373
44 0.560000  0.880800  0.880800
45 0.580000  0.874190  0.874190
46 0.600000  0.867554  0.867554
```

C++ SOURCE B

```
 1 #include <stdio.h>
 2 #include "mxyzptlk.rsc"
 3
 4 // ..... Globals .....................................
 5 const double tolerance = 1.0e-8;
 6 const int    dfNdx[]   = { 0, 1 };
 7 const int    maxIter   = 15;
 8
 9 double  m;
10 DA      G, md, xd;
11
12 // ..... Functions ...................................
13 DA     F( DA& x )    { return ( x - cos( md*x ) ); }
14 double F( double x ) { return ( x - cos( m *x ) ); }
15
16 double solve( double m, double x ) {
17 double xOld, s[2];
18 int    i;
19
20 md.setVariable( m, 0 );
21 xd.setVariable( x, 1 );
22 md.fixReference();
23 xd.fixReference();
24
25 s[0] = m;    i = 0;
26 s[1] = x;    xOld = 123456789.0;
27
28 while( ( tolerance < fabs( x - xOld ) )
29    &&  ( i++ < maxIter )
30      ) {
31   xOld = x;
32   G = F( xd );
33   G = xd - ( G / G.D( dfNdx ) ); // Sets up Newton's algorithm.
34   x = G.multiEval( s );          // Recursive step.
35   md.setVariable( m, 0 );
36   xd.setVariable( x, 1 );
37   md.fixReference();
38   xd.fixReference();
39   s[1] = x;
40 }
41 G = F( xd );                    // This is done to assure that G has
42 G = xd - ( G / G.D( dfNdx ) );  // a reference point identical to
43                                 // md and xd, at least at the start.
44 return x;
45 }
46
47 // ..... Main program ..................................
48 main() {
49 const int  dim       = 2;
```

```
50 const int  maxWeight = 5;
51 double     x;
52 double     mLo, mHi, step, s[2];
53 int        i, j;
54
55 DASetup( dim, maxWeight, dim ); // This form required
56 DA sd[2], Y;                    // to do multiEval( DA* )
57
58 printf("Enter the value of m\n");
59 printf("and an initial guess for x: ");
60 scanf("%lf %lf", &m, &x );
61 getchar();
62 printf( "You have chosen m = %lf and x = %lf\n", m, x );
63
64 // ...... First solve for the value of x(m) ................
65 x = solve( m, x );
66 printf( "x( %lf ) = %lf   ", m, x );
67 printf( "Defect is: %lf\n", F(x) );
68
69 // ...... Now calculate the derivatives ....................
70 sd[0] = md;
71 sd[1] = xd;
72 for( i = 0; i < maxIter; i++ ) {
73   xd = G.multiEval( sd );
74   sd[0] = md;
75   sd[1] = xd;
76 }
77
78 xd.peekAt();
79
80 // ...... And validate .....................................
81 printf( "\nThe real test:\n" );
82 printf( "Enter mLo, mHi, and step: " );
83 scanf("%lf %lf %lf", &mLo, &mHi, &step );
84 getchar();
85
86 Y = xd;          // xd is a global variable, and solve() will
87                  // change its value.
88 s[1] = 1.0;      // The value of s[1] no longer matters.
89 for( m = mLo; m < (mHi + 0.5*step); m+=step ){
90   x = solve( m, x );
91   s[0] = m;
92   printf( "%lf  %lf  %lf  %lf\n",
93           m,   x,   Y.multiEval( s ),   F(x)
94         );
95 }
96
97 } // End function: main()
```

OUTPUT B:

```
 1 Enter the value of m
 2 and an initial guess for x: 0.3 0.6
 3 You have chosen m = 0.300000 and x = 0.600000
 4 x( 0.300000 ) = 0.958907   Defect is: 0.000000
 5
 6 Count  = 6, Weight = 5, Max accurate weight = 5
 7 Reference point:
 8 3.000000e-01  9.589070e-01
 9 Index:  0  0   Value:  9.589070e-01
10 Index:  1  0   Value: -2.507213e-01
11 Index:  2  0   Value: -2.794859e-01
12 Index:  3  0   Value:  3.644146e-01
13 Index:  4  0   Value: -2.356161e-02
14 Index:  5  0   Value: -3.628218e-01
15
16 The real test:
17 Enter mLo, mHi, and step: 0.1 0.5 0.1
18 0.100000  0.995053  0.995035   0.000000
19 0.200000  0.980821  0.980821   0.000000
20 0.300000  0.958907  0.958907  -0.000000
21 0.400000  0.931399  0.931398  -0.000000
22 0.500000  0.900367  0.900345   0.000000
```